An Evolutionary Path to Object Storage Access

David Goodell,* Seong Jo Kim,† Robert Latham,*
Mahmut Kandemir,† and Robert Ross*
*Mathematics and Computer Science Division
Argonne National Laboratory
{goodell,robl,rross}@mcs.anl.gov
†Department of Computer Science and Engineering
Pennsylvania State University
{seokim,kandemir}@cse.psu.edu

***Abstract*—High-performance computing (HPC) storage systems typically consist of an object storage system that is accessed via the POSIX file interface. However, rapid increases in system scales and storage system complexity have uncovered a number of limitations in this model. In particular, applications and libraries are limited in their ability to partition data into units with independent concurrency control, and mapping complex science data models into the POSIX file model is inconvenient at best.**

In this paper we propose an alternative interface for use by applications and libraries that provides direct access to underlying storage objects. This model allows applications and libraries to organize storage access around these objects in order to avoid lock contention without needing to create many separate files. Additionally, complex data models are more readily organized into multiple object data streams, simplifying the storage of variable-length data and allowing a choice of degree of parallelism related to access needs. Our approach provides for datasets stored in this new model to coexist with POSIX files, allowing evolution to the new model over time. We apply these concepts in the PVFS, PLFS, and Parallel netCDF packages to prototype the model and describe our experiences.

I. Introduction

Computing systems have long presented persistent storage as a collection of files organized in a directory hierarchy. The POSIX file system interface is a standard for accessing persistent storage in this model [1], and both the standard and the "files and directories" abstraction have been highly successful. Various alternatives and extensions to this model have existed over time; as parallel computing has grown, groups have investigated various options for parallelism in the I/O path that expose alternative file system models. The MPI-IO portion of the MPI-2 standard [2] defined an interface for accessing POSIX-style files from MPI applications. While this interface has seen limited success, alternative file models have not gained traction in the parallel I/O context.

Many parallel file systems (PFSes) today [3], [4], [5], [6] support the *object storage model*, where a file is mapped onto a set of objects distributed across storage servers. These objects can each be thought of as an independent linear array of bytes, but PFSes combine these into a single linear array of bytes via some distribution function (Figure 1). This approach often leads to poor performance [7] and does not directly support the complicated datasets of many scientific applications that instead use libraries such as Parallel netCDF [8] and HDF5 [9] to better capture structure.

As the community builds and deploys ever more capable computing systems, the demands placed on the storage system grow at a rate commensurate with the ability to generate new results. Researchers and developers are examining all aspects of the storage stack in order to maximize efficiency of the system, from new methods of organizing I/O among compute processes [10], [11], to the addition of new architectural features that better balance I/O traffic over time [12], [13]. However, the majority of this work assumes an external storage system providing the POSIX file model.

In this work we present an extension to the POSIX API that exposes a different abstraction for storing data that allows the user (i.e., library or application) to take advantage of the multiple byte arrays already present in arrays of storage objects (Section III). This extended model coexists with traditional POSIX file data, providing a clean path for adoption of the new model by applications and libraries over time. We then describe an implementation of this extension using PVFS [4] as our starting point (Section IV) and discuss how we have modified two popular I/O libraries, parallel log-structured file system (PLFS) [11] and Parallel netCDF [8], to take advantage of the extension's features (Section V).

II. Related Work

A number of approaches have extended the byte stream model for files. The Microsoft NTFS file system includes the concept of *alternate data streams*, which are a method of associating multiple, named "streams" of data under the same file name [14]. The IBM Virtual Storage Access Method (VSAM) of persistent data access allows for multiple access modes (sequential, random, and indexed), the definition of records of fixed or variable size, and a concept of a key associated with each record [15].

The Galley parallel file system [16] supported the concept of *subfiles*. Rather than supporting a single linear array of bytes in a file, N subfiles were allocated at creation time, each mapping to a distinct disk. These subfiles held a set of *forks*, each of which provided a linear array of bytes. Access was provided on a per fork basis, meaning that any distribution of data across multiple subfiles had to be managed by upper software layers. Nieuwejaar and Kotz [16] note that these forks

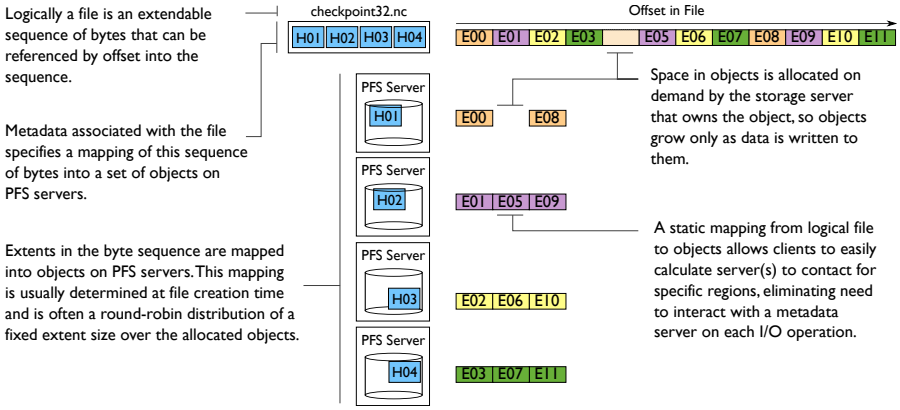


Fig. 1. Distribution of POSIX file byte array into a set of objects. POSIX model means striping must happen without regard to any application structure.

are lighter weight from a metadata perspective than separate files and provide a convenient means of grouping related data together.

More recently, researchers have investigated similar methods for guiding I/O from many processes to distinct subfiles (reusing the name) on Lustre [17]. In this approach, subfiles are created as applications write data, avoiding the need for creating a file per process and the associated overheads. These subfiles can span multiple storage nodes and are stitched back into a single stream of bytes ("joined") at file close time, so that after file close the data appears as a single POSIX file.

The Distributed Application Object Storage (DAOS) [18] model being developed by Intel, EMC, and the HDF Group has similarities to the model described here, though it departs more aggressively from POSIX than our work does. The DAOS model, for example, employs a separate scalable object namespace and defines transactions over collections of clients.

III. Objects in a POSIX Namespace

Many (if not most) PFSes today use an object storage abstraction [19] underneath. Figure 1 depicts an example of how these systems operate: the file system uses metadata information associated with the file name that includes a list of objects and a set of distribution parameters to map from offsets in the POSIX byte array (i.e., offsets in the file) to regions of objects on specific storage servers.

The major difference from the Galley [16] model is that in modern PFSes, the PFS, rather than the user (or a library), is responsible for performing the mapping from POSIX file locations to the byte arrays in storage.

Our goal in this work is to provide a new abstraction for storage that enables higher performance for HPC applications while coexisting with the POSIX name space that is the basis of most HPC storage deployments. Our approach is to expose a set of objects (actually an ordered list) associated with a single file name (a *container*), leveraging the fact that this reflects the underlying storage organization of many systems today. This model moves the responsibility of mapping application data structures into the objects from the file system to the libraries or application. In this work we assume that the underlying storage performs consistency management (i.e., locking), if any, on a per object basis; we assume that creating many objects under a single file name is faster than creating multiple files in the name space. This approach has two major advantages. First, the model separates the creation of multiple data streams from the creation of names in the name space. Second, the model allows the multiple data streams present in the individual objects to be used directly for organizational purposes. These advantages are showcased in Section V.

IV. Supporting Object Access in PVFS

To prototype our extensions, we modified the Parallel Virtual File System source obtained from CVS in May 2012 (v2.8.2 with additions). Only client-side modifications were required to facilitate the new model. A new API for object-based access was added and implemented by using code derived from the existing PVFS2 POSIX code.

A. New API

The prototype API provides all essential functionality through a simple API. Listing 1 shows the entire API.

The `tr_create_objs` routine creates a new collection, referenced by `path_name`, with `num_objs` number of objects. In the prototype, this path corresponds to a PVFS file that can be seen via regular `ls`. The object IDs are returned in the `ids` array. An optional *constraint* interface allows callers to specify key-value pairs that constrain object placement and/or performance characteristics. The constraints are currently unused in the prototype but are provided as an avenue for future exploration.

In order to obtain the ordered list of object IDs for an existing collection, `tr_lookup_objs` may be called. It takes arguments similarly to `create`, but with the small difference of allocating the ID array instead of requiring it to be allocated ahead of time and then populated. The number of objects in the named collection may not be known a priori, so this information is also returned by this call.

Data can be read from an object or written to an object with one of the four `tr_obj_` I/O calls: `read_contig`,

write_contig, readx, and writex. The first two routines permit simple contiguous I/O, while the last two offer a noncontiguous-I/O interface. Their definitions are intuitive and correspond roughly to POSIX read/write and readx/writex.

B. PVFS2 Client Implementation Details

The existing, underlying PVFS2 object model decomposes a logical (user-visible) POSIX file into a single *metafile*, containing metadata, and one or more *datafiles*, containing file extent data. A distribution function is then applied to map logical file extents into extents within the datafiles. The datafiles are identified by a PVFS_object_ref, which is a (handle, file-system-id) integer pair with 128-bit total width.

Our prototype reuses these existing concepts, although the significance of the metafile is somewhat reduced, since the distribution function and parameters are no longer consulted. The primary use of the metafile is simply to map from a PVFS pathname to a set of constituent objects. The tr_oid type is merely a PVFS_object_ref, allowing direct I/O access by a client to/from individual objects without intervening metadata lookups.

PVFS2 uses a code generation process to generate "state machine" code from a C-based domain language. These state machines simplify the implementation of otherwise tedious nonblocking I/O code paths. Two new state machines were added to the PVFS2 prototype: one for object collection creation and one for read/write I/O operations to a single object. The creation state machine is loosely based on the POSIX file creation state machine (sys-create.sm), with numerous simplifications. In particular, the "file stuffing" optimizations that defer datafile creation [20] were eliminated in order to ensure that individual object I/O paths would be straightforward.

The object read-write state machine is based on the POSIX I/O state machine (sys-io.sm). Since file stuffing has been disabled and this code accesses only individual datafiles, rather than high-level logical files, querying the distribution function is not needed.

V. Using Objects in HPC I/O Libraries

Increasingly, applications are taking advantage of I/O libraries that provide either higher performance (e.g., PLFS) or a more convenient abstraction than POSIX for storing scientific data (e.g., HDF5, PnetCDF, ADIOS), or both. In this section, we consider how our new API affects these libraries.

A. Parallel Log-Structured File System

Log-structured file systems [21] are designed to achieve high write performance. All updates to data and metadata are sequentially written to a contiguous stream, called a log. Similarly, the Parallel Log-structured File System (*PLFS*) was designed to improve checkpoint bandwidth for writes. PLFS is implemented as a user-space file system, exposed through FUSE [22] or MPI-IO [2]. In Figure 2(a) when multiple client processes all open/write the shared checkpoint file ("foo") in the PLFS layer, PLFS creates a *container* in the underlying target PFS (e.g., Lustre, PanFS). Here, the container is a hierarchical directory tree that consists of a single top-level directory (i.e., "foo/") with the same name as the logical file and subdirectories (i.e., "hostA/" and "hostB/"). On each compute node a unique subdirectory is created for each client host, and a pair of *data* and *index* file (i.e., "data.#" and "index.#") is created for each process within those directories. Though multiple processes open the same logical file for writing and share the top-level container, each process opens its unique data file within the subcontainer and appends writes to it in a log-structured fashion. After writing to a data file, PLFS appends the metadata information to the associated index file maintained to permit later reads. By remapping writes to a shared checkpoint file from a parallel application to a nonshared data file, PLFS converts an N-1 strided access pattern into an N-N one.

Our object storage model is illustrated in Figure 2(b). For our prototype, we plugged the ad_plfs interface into the ROMIO ADIO layer of MPICH2-1.5, porting the version of PLFS currently available online. In our current implementation, application programs directly make MPI-IO calls to reach PLFS. Also, the PLFS library is modified to support our new API for object-based access, described in Section IV. Unlike the container in stock PLFS, a hierarchical directory tree, the container ("foo") here is an object collection represented by a single file name on the underlying parallel file system. The collection consists of (index, data) object pairs for each process. When the application writes a checkpoint file, the root process creates the container with (index, data) object pairs for each processes and broadcasts the constituent object IDs to the other processes. Then, each process obtains its corresponding object ID pairs from the received object IDs. During writes, each process appends data to the data object and metadata information to the associated index object using two calls to tr_obj_write_contig.

When reading a file, for example, restarting a computation from a checkpoint file generated in a previous run, the root process looks up the object IDs and broadcasts them to the others. Index reconstruction is largely unchanged from the original PLFS approach; it just occurs on multiple objects rather than multiple files. Any optimizations the PLFS team might develop related to index storage should be easily applied in this new approach.

B. Parallel netCDF

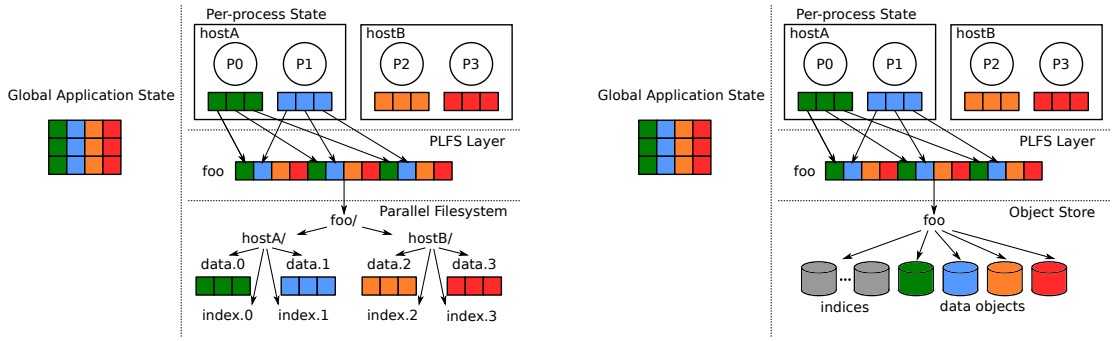
The Parallel netCDF (*PnetCDF*) library provides an interface for parallel reading and writing of data in the netCDF [23] file format. NetCDF allows for the storage of multiple, typed, multidimensional arrays in a single dataset (i.e., file) along with attributes on the arrays and the dataset itself. Arrays can be of fixed dimensions (termed *nonrecord arrays*) or have one dimension in which they may grow (*record arrays*). When mapped into a POSIX file, tiles of these record arrays are interleaved in the file so that space may be allocated algorithmically as the record arrays grow. In general, the

```
typedef struct { int n; char **keys; char **vals; } tr_constraints;

int tr_create_objs(const char *path_name, tr_constraints *constraints,
                   int num_objs, tr_oid ids[]);
int tr_destroy_objs(const char *path_name);
int tr_lookup_objs(const char *path_name, int *num_objs, tr_oid *ids[]);

int tr_obj_read_contig(tr_oid id, tr_offset offset, size_t count, void *buffer, size_t *nout);
int tr_obj_write_contig(tr_oid id, tr_offset offset, size_t count, const void *buffer);
ssize_t tr_obj_readx(tr_oid id, const struct iovec *iov, size_t iov_count,
                     const struct xtvec *xtv, size_t xtv_count, size_t *nout);
ssize_t tr_obj_writex(tr_oid id, const struct iovec *iov, size_t iov_count,
                      struct xtvec *xtv, size_t xtv_count);
```

Listing 1. Object API



(a) Data rearrangement in PLFS.

(b) Data arrangement in our object storage model.

Fig. 2. Comparisons of data remapping schemes in PLFS and the object storage model. In (a), PLFS creates a container (a hierarchical directory tree) and appends data and index to the corresponding files. In the object storage model in (b), a container is created as a file containing (index,data) object pairs for each process. Data and metadata are appended to the corresponding object, respectively.

ability to grow arrays over time is desired by application teams, and support for this model is provided in HDF5 and has been explored in other research activities [24].

Figure 3(a) depicts the mapping of a dataset described by netCDF into a POSIX file. Header data (shown) and nonrecord arrays (not shown) come first in the POSIX file's byte stream. The two record arrays in the dataset are interleaved at the end of the file. This logical flat file is then distributed among servers in the PFS, without regard for compatibility between file system distribution parameters and the layout of the netCDF arrays. For example, a record array with an 8 MB record size (e.g., a $100 \times 100 \times 100$ 3-D array of `doubles`) might be interleaved with another record array with a 24 MB record size, all of which is then striped across 12 servers with a 64 KiB stripe size. The factorizations and multiplicities of these sizes lead to irregularly aligned access. The presence of the header has further potential to misalign data, although the header may be padded to a stripe boundary (if the stripe unit is known).

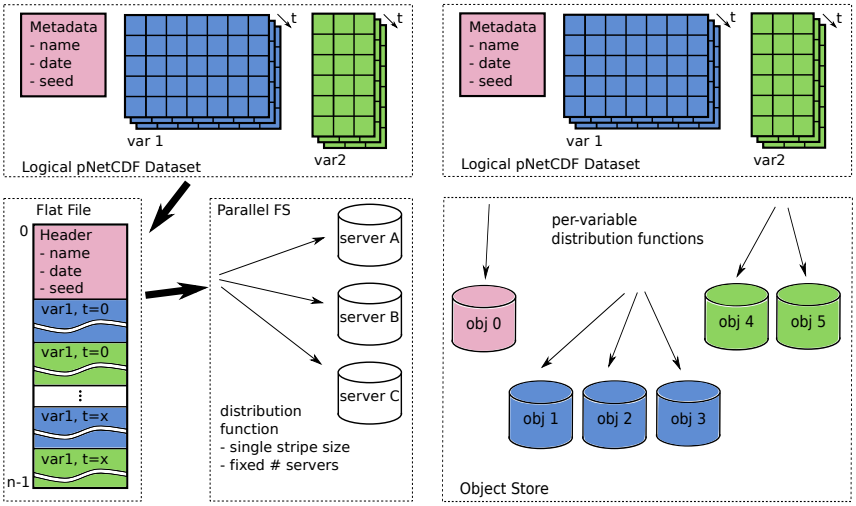
Record variable storage has another performance drawback. The interleaved storage approach results in pathologically noncontiguous accesses for several common access methods. For example, reading a subcube out of a 3D array stored in record format can result in reading vastly more data than needed because of "data sieving" in the underlying MPI-IO library [25].

Starting with Parallel netCDF SVN revision r1049, we modified the library to map netCDF datasets onto our new abstraction. Figure 3(b) shows how our object-based PnetCDF prototype maps the same dataset onto the set of objects provided by our object storage model. The header and each array are mapped to their own set of one or more objects. One minor immediate benefit is that doing so greatly simplifies the implementation effort involved in reading/writing from/to variables, especially for noncontiguous access. However, a much more valuable product of this shift is the ability of PnetCDF to control the data distribution on a per variable basis. This makes it substantially easier to avoid misaligned data access and to reason about parallel I/O performance, all on a per dataset basis, rather than a system wide basis.

In our prototype, each PnetCDF variable has its own distribution function, described by a (stripe_size_bytes, object_set) pair. Data is striped byte wise in a row-major fashion (bytes from the fastest-varying dimension are adjacent). More complex distributions could be implemented easily, if needed. By default, arbitrary parameters of two objects and a stripe size of 64 KiB are used. In the future we intend to examine heuristics for automatically selecting object counts and stripe sizes based on the array layout described by the user.

VI. Other Considerations

File size. Our approach moves the role of the distribution function into applicaiton or library space. Without a distribution function, the PFS must report file size (as when `stat` is called) by simply computing the total size of data stored



(a) Over a Typical Parallel File System (b) Over an Object-based Storage System

Fig. 3. Parallel netCDF data layout over two storage models. The data distribution in (a) is not under the control of the PnetCDF library. The scheme in (b) exposes this choice to the I/O library, enabling access to maximal performance and efficiency from the underlying storage system while simultaneously simplifying the implementation of PnetCDF.

in constituent objects. Such an approach may not deal with "sparse files" accurately. Furthermore, if the PFS could expose the size of individual objects, some application structure could be visible with traitional POSIX utilities.

Access control and extended attributes. We propose that access control and extended attrbutes, two pieces of POSIX functionality, be unchanged in our model. Existing metadata management should be adequate, and fine-grained control over access to individual objects in the container does not appear to be needed for the use cases that we have studied to date.

Copying collections. Notionally, a collection could be copied by creating a new set of objects of the same size as the collection of objects in the source and copying the contents of each object into the corresponding object in the new list. However, we do not assume the OIDs would be identical in the new collection. If libraries or applications using this model indirectly reference object IDs internally (i.e., store an index into the object list when referring to the object holding some variable, rather than the OID itself), this approach is adequate, although it requires a custom utility. This approach implies the need to hold portions of the OID list in memory for referencing. For lists of small size, this is not considered a problem; if these lists grow larger, a more capable replacement for `tr_lookup_objs` will be needed to facilitate reading subsets of the overall list.

VII. Conclusions and Future Work

In this paper, we have presented a new abstraction for storage that enables higher performance for HPC applications while coexisting with the legacy POSIX name space. Compared with the POSIX interface, this container of objects model maps more closely both to application and library needs and to the architecture of modern storage systems. By moving the responsibility of mapping application data structures into storage objects out of the storage system, we give the application greater control over performance (e.g., from alignment and metadata issues) while simultaneously offering opportunities for simpler implementation through simpler data organization. We implemented a prototype of this model in PVFS and evaluated the qualitative impact of adapting PLFS and PnetCDF to use this prototype. We found that our model fits naturally into these I/O libraries, generally simplifying their implementation. We showed how this new model can coexist with the POSIX model, aiding transition away from POSIX. Furthermore, the fact that our model is particularly suited to I/O middleware libraries should even further ease the transition for applications in the future.

We intend to study our model in scenarios with a far greater number of objects. For example, rather than mapping an array in a PnetCDF dataset to a handful of objects, is there value in mapping to thousands of objects? On the flip-side of this question is an exploration of the design space for the storage system itself. This paper has focused primarily on the impact on libraries, as well as a straightforward implementation of our new model in PVFS. With many more objects, the storage system may have greater opportunity for internal optimization. Another area of future work is to study the impact of object locality and how that might best be expressed to an application by the storage system. Object storage may also affect the way that I/O forwarding [26] stacks are designed and implemented.

Acknowledgments

This work was supported by the U.S. Department of Energy, under Contract DE-AC02-06CH11357. We thank Lee Ward and Eric Barton for participating in discussions that have helped guide us to this model.

REFERENCES

[1] IEEE, *2004 (ISO/IEC) [IEEE/ANSI Std 1003.1, 2004 Edition] Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application: Program Interface (API) [C Language]*. New York, NY USA: IEEE, 2004.

[2] Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface," July 1997, `http://www.mpi-forum.org/docs/docs.html`.

[3] P. J. Braam, "The lustre storage architecture," Cluster File Systems, Inc., Tech. Rep., 2003. [Online]. Available: http://lustre.org/docs/lustre.pdf

[4] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for linux clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*. Atlanta, GA: USENIX Association, Oct. 2000, pp. 317–327.

[5] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the Panasas parallel file system," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008, pp. 17–33.

[6] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006, pp. 307–320.

[7] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and improving computational science storage access through continuous characterization," *Trans. Storage*, vol. 7, no. 3, pp. 8:1–8:26, Oct. 2011. [Online]. Available: http://doi.acm.org/10.1145/2027066.2027068

[8] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, , and M. Zingale, "Parallel netCDF: A high-performance scientific I/O interface," in *Proceedings of SC2003*, Nov. 2003.

[9] "HDF5," `http://www.hdfgroup.org/HDF5/`.

[10] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and integration for scientific codes through the adaptable IO system(ADIOS)," in *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments (CLADE '08)*, 2008, pp. 15–24.

[11] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A checkpoint filesystem for parallel applications," in *Proceedings of SC09*, Nov. 2009.

[12] P. Nowoczynski, N. Stone, J. Yanovich, and J. Sommerfield, "Zest checkpoint storage system for large supercomputers," in *Petascale Data Storage Workshop, 2008. PDSW '08. 3rd*, November 2008, pp. 1–5.

[13] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *Proceedings of the 2012 IEEE Conference on Massive Data Storage*, Pacific Grove, CA, Apr. 2012.

[14] H. Berghel and N. Brajkovska, "Wading into alternate data streams," *Commun. ACM*, vol. 47, no. 4, pp. 21–27, Apr. 2004. [Online]. Available: http://doi.acm.org/10.1145/975817.975836

[15] D. Lovelace, R. Ayyar, A. Sala, and V. Sokal, "VSAM demystified (second edition)," IBM International Technical Support Organization, Tech. Rep. SG24-6105-01, Sep. 2003.

[16] N. Nieuwejaar and D. Kotz, "The Galley parallel file system," *Parallel Computing*, vol. 23, no. 4, pp. 447–476, June 1997. [Online]. Available: http://www.cs.dartmouth.edu/~dfk/papers/nieuwejaar:jgalley.ps.gz

[17] W. Yu, J. Vetter, R. S. Canon, and S. Jiang, "Exploiting lustre file joining for effective collective IO," in *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007)*, May 2007.

[18] E. Barton, "DAOS containers: A storage abstraction for exascale," presentation at Argonne National Laboratory, Oct. 2011.

[19] R. O. Weber, "SCSI object-based storage device Commands-2 (OSD-2), revision 05a," INCITS Technical Committee T10/1729-D working draft, work in progress, Tech. Rep., Jan. 2009.

[20] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig, "Small-file access in parallel file systems," in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, Apr. 2009.

[21] M. Rosenblum and J. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992.

[22] "Filesystem in userspace." [Online]. Available: http://fuse.sourceforge.net/

[23] R. Rew and G. Davis, "The unidata netCDF: Software for scientific data access," in *Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography and Hydrology*, Feb. 1990, pp. 33–40.

[24] E. Otoo and T. Merrett, "A storage scheme for extendible arrays," *Computing*, vol. 31, pp. 1–9, 1983, 10.1007/BF02247933. [Online]. Available: http://dx.doi.org/10.1007/BF02247933

[25] T. Peterka, H. Yu, R. Ross, K.-L. Ma, and R. Latham, "End-to-end study of parallel volume rendering on the IBM Blue Gene/P," in *Proceedings of ICPP 09*, Vienna, Austria, Sep. 2009.

[26] K. Ohta, D. Kimpe, J. Cope, K. Iskra, R. Ross, and Y. Ishikawa, "Optimization techniques at the I/O forwarding layer," in *Proceedings of the IEEE International Conference on Cluster Computing*, Sep. 2010.